CHAPTER 5: POINTERS AND ARRAYS

A pointer is a variable that contains the address of another object. Pointers are very much used in C, partly because they are sometimes the only way to express some computation, and partly because they usually lead to more compact and efficient code than can be obtained in other ways.

Pointers have been lumped with the **goto** statement as a marvelous way to create impossible-to-understand programs. This is certainly true for careless programming, and it is easy to create pointers that point somewhere unexpected. With discipline, however, pointers can also be used to achieve clarity and simplicity. This is the aspect that we will try to illustrate.

5.1 Pointers and Addresses

Since a pointer contains the address of an object, it is possible to access the object "indirectly" through the pointer. Suppose that x is a variable, say an int, and that px is a pointer, created in some as yet unspecified way. Then the variable px can be set to the *address* of x by the assignment

```
px = &x;
```

The unary operator & gives the address of an object (it can be applied only to variables and functions; constructs like &(x+1) and &3 are illegal). So now px contains the address of x.

The unary operator * is the inverse of &; it treats its operand as a variable that contains the address of the ultimate target, and accesses indirectly through that address to fetch the contents. Thus if y is also an int,

```
y = *px;
```

assigns to y the contents of whatever px points to. So the sequence

```
px = &x;
y = *px;
```

is equivalent to

```
y = x;
```

It is also necessary to declare the variables participating in all of this:

```
int x, y;
int *px;
```

The declaration of x and y is what we've seen all along. The declaration of the pointer px is new.

```
int *px;
```

is intended as a mnemonic; it says that the combination *px is an int, that is, if px occurs in the context *px, it is equivalent to a variable of type int. This reasoning is useful in all cases involving complicated declarations. It is quite analogous to a declaration like

```
float atof( );
```

which says that atof( ) is an object of type float.

You should also note the implication in the declaration that a pointer is constrained to point to a particular kind of object. It is fraught with peril to declare a variable to be a pointer to one type, then use it as a pointer to another.

Pointers can occur in expressions. For example, if px points to the integer x, then *px can occur in any context where x could.

```
y = *px + 1
```

sets y to 1 more than x;

```
printf("%d\n", *px);
```

prints the current value of x; and

```
d = sqrt( (double) *px);
```

produces in d the square root of x, which has to be coerced into a double before being passed to sqrt.

In expressions like

```
y = *px + 1
```

* and & bind more tightly than arithmetic operators, so this adds 1 to whatever px points at, and assigns it to y. We will return shortly to the meaning of

```
y = *(px+1)
```

Pointer references can also occur on the left side of assignments.

```
*px = 0;
```

sets x zero, and

```
*px += 1
```

increments it, as does

```
(*px)++
```

The parentheses are necessary here; without them, the expression would

increment px instead of what it points to, because unary operators like * and ++ are evaluated right to left.

Finally, since pointers are variables, they can be manipulated as other variables can. If py is another pointer to int, then

```
py = px
```

has the obvious meaning of making py point to whatever px points to.

5.2 Pointers and Function Arguments

Since C passes arguments to functions by "call by value", there is no direct way for the called function to alter a variable in the calling function. What do you do if you really have to change an ordinary argument? For example, a sorting routine might exchange two out-of-order elements with a function called swap. It is not enough to write

```
swap(x, y)    /* WRONG */
int x, y;
{
        int temp;

        temp = x;
        x = y;
        y = temp;
}
```

because swap *can't* affect the arguments x and y. Fortunately, pointers provide a way to obtain the desired effect. In the calling program, pass not the argument but a pointer to it:

```
swap(&a, &b);
```

Since the operator & gives the *address* of a variable, &a is a pointer to a.

In swap itself, declare the arguments to be pointers, and treat them as such:

```
swap(px, py) /* interchange *px with *py */
int *px, *py;
{
        int temp;

        temp = *px;
        *px = *py;
        *py = temp;
}
```

One common use of pointer arguments is in functions that must return more than a single value. (You might say that swap returns two values, the new values of its arguments.) As an example, a function which reads input and

converts it to a stream of integers has to return the value it found, and also some indication of whether or not end of file has occurred. And these values have to be returned in different places, for no matter what value is used for EOF, that could also be some legitimate integer from the input.

One solution, which is based on the input function **scanf** that we will talk about in Chapter 7, is to write a function **getint** which returns 1 if it really found a number, 0 if it found no number, and −1 if it found end of file. The actual numeric value is returned in the argument, which then must be a pointer to an integer. In this way end of file and error signals can be distinguished from legal numeric values.

The call

```
int n, stat;
...
stat = getint(&n);
```

sets n to the next integer found in the input and **stat** to the status returned by **getint**.

```
getint(pn)    /* get next integer from input */
int *pn;
{
        int c, sign = 1;

        while ((c = getchar( )) == ' ' || c == '\n' || c == '\t')
                ;       /* skip white space */
        if (c == '+' || c == '-') {       /* sign */
                sign = (c=='+') ? 1 : -1;
                c = getchar( );
        }
        for (*pn = 0; c >= '0' && c <= '9'; c = getchar( ))       /* integer part */
                *pn = 10 * *pn + c - '0';
        *pn *= sign;
        if (c == ' ' || c == '\n' || c == '\t')
                return(1);    /* normal integer found */
        else if (c == EOF)
                return(-1);
        else
                return(0);    /* some error */
}
```

Throughout **getint**, *pn is used as an ordinary **int** variable.

Exercise 5-1: Write **getfloat**, the floating point analog of **getint**. What type does **getfloat** return as its function value? □

5.3 Pointers and Arrays

There is a strong relationship between pointers and arrays, strong enough that pointers and arrays really should be treated simultaneously. Any operation which can be achieved by array subscripting can also be done with pointers. The pointer version will in general be rather more efficient but, at least to the uninitiated, somewhat harder to grasp immediately.

The declaration

```
int a[10]
```

defines an array **a** of size 10, that is a block of 10 consecutive objects named **a[0]**, **a[1]**, ..., **a[9]**. The notation **a[i]** means the i-th element of the array, that is, i positions from the beginning. If **pa** is a pointer to **int**, declared as

```
int *pa
```

then the assignment

```
pa = &a[0]
```

sets **pa** to point to the zeroth element of **a**. Now the assignment

```
x = *pa
```

copies **a[0]** into **x**.

If **pa** points to a particular element of the array **a**, then *by definition* **pa + 1** points to the next element, and in general **pa ± i** points i elements before or after **pa**. Thus, if **pa** points to **a[0]**,

```
*(pa+1)
```

refers to the contents of **a[1]**, and **pa + i** is the address of **a[i]**.

The definition of "adding 1 to a pointer," and by extension, all pointer arithmetic, is that the increment is scaled by the size in storage of the object that is pointed to. This makes the correspondence between indexing and pointer arithmetic very close.

In fact, the association is even closer. Again by definition, a reference to an array of a particular type is converted by the compiler to a pointer to the beginning of the array. In effect, an array name *is* a pointer expression. This has quite a few useful implications.

Since the name of an array is a synonym for the location of the zeroth element, the assignment

```
pa = &a[0]
```

can also be written as

```
pa = a
```

The array name **a**, being of type "array", is a pointer expression, and can be assigned to a pointer of the same type.

Rather more surprising, at least at first sight, is the fact that a reference to **a[i]** can also be written as ***(a+i)**. By definition, in evaluating **a[i]**, C

converts it to *(a+i) immediately; the two forms are identical. Applying the operator & to both parts of this equivalence, it follows that &a[i] and a+i are also identical. a+i is the address of an object i elements beyond a.

There is one difference between an array name and a pointer that should be kept in mind. A pointer is a variable, so pa=0 or pa++ make perfect sense. But an array name is a name, not a variable: constructions like a=0 or a++ are illegal.

There is an exception to this rule, an important one. Within a function, C interprets arguments which are declared "array of ..." as actually being "pointer to ...", so that either subscripting or indexing can be used. When an array name is passed to a function, the function can at its convenience believe that it has been handed either an array or a pointer, and manipulate it accordingly. It can even use both kinds of operations if it seems appropriate and clear.

As a consequence of this conversion from array to pointer, it is possible to pass a part of an array to a function. For example,

```
f(&a[2])
```

and

```
f(a+2)
```

both pass to the function f the address of element a[2]. Within f, the declaration can read

```
f(arr)
int arr[ ];
{
        ...
}
```

or

```
f(arr)
int *arr;
{
        ...
}
```

So as far as f is concerned, the argument either an array or a pointer, and the fact that it is really part of a larger array is of no consequence.

5.4 Address Arithmetic

We have seen that p++ increments p by 1 to point to the next element of whatever kind of object p points to, and p += i increments p to point i elements beyond where it currently does. These and similar constructions are among the simplest and most common forms of pointer or address arithmetic; many others are also possible.

C is quite consistent and regular in its approach to address arithmetic. To illustrate some of its properties, let us build a rudimentary storage allocator (but useful in spite of its simplicity). There are two routines. alloc(n) returns a pointer to n consecutive character positions, which can be used by the receiver of the pointer for storing any characters it likes. free(p) releases the storage thus acquired so it can be later re-used. The routines are "rudimentary" because the calls to free must be made in the opposite order to the calls made on alloc. That is, the storage managed by alloc and free is a stack, or last-in, first-out queue. The standard C library provides an alloc and free which have no such restrictions. In the meantime, many applications really only need a trivial alloc to hand out little pieces of storage of unpredictable sizes at unpredictable times.

The simplest implementation is to have alloc and free hand out pieces of a large character array which we will call alloc_buf. This array is private to alloc and free; since they deal in pointers, not array indices, no other routine need know the name of the array, which can be declared external static. In fact, in practical implementations, the array may well not even have a name; rather it might be obtained by asking the operating system for a pointer to some unnamed free storage. (This is one place where pointers can serve and array indices can't.)

The other piece of data needed is some record of how much of alloc_buf has been used. A pointer to the next free element is convenient; we will call it alloc_p.

When alloc is asked for n bytes, it checks to see if there is enough room, and if so returns the current value of alloc_p, then increments it by n. free(p) merely sets alloc_p to p after a bit of error checking.

```
#define     NULL 0      /* illegal pointer value for error reporting */
#define     ALLOC       1000  /* size of available space */

static char   alloc_buf[ALLOC];  /* storage for alloc and free */
static char   *alloc_p      = alloc_buf; /* next free position */

char *alloc(n)      /* return pointer to n bytes */
int n;
{
        if (alloc_p + n <= alloc_buf + ALLOC)       /* it fits */
                return((alloc_p += n) - n);
        else            /* not enough room */
                return(NULL);
}

free(p)         /* free storage for alloc */
char *p;
{
        if (p >= alloc_buf && p < alloc_buf + ALLOC)
                alloc_p = p;
}
```

Some explanations. In general a pointer can be initialized just as any other variable, though normally the only meaningful values are NULL or some expression involving previously defined pointers of the same type. The statement

```
static char   *alloc_p      = alloc_buf; /* next free position */
```

defines alloc_p to be a character pointer and initializes it to point to alloc_buf, which is the next free position when the program starts. This could have also been written

```
char   *alloc_p      = &alloc_buf[0];
```

but since the array name *is* the address of the zeroth element, it's redundant to do so.

The test

```
if (alloc_p + n <= alloc_buf + ALLOC)       /* it fits */
```

checks if there's enough room to satisfy this request. If there is, the new value of alloc_p would be at most one beyond the end of alloc_buf. If the request can be satisfied, alloc returns a normal pointer (notice the declaration of the function itself). If not, alloc must return some signal that an error has happened. C guarantees that a pointer that really points at data will never be zero, so returning zero signals some abnormal event, in this case, no space. We write NULL instead of zero, however, to indicate more clearly that this is a special value for a pointer.

Tests like

```
if (alloc_p + n <= alloc_buf + ALLOC)        /* it fits */
```

and

```
if (p >= alloc_buf && p < alloc_buf + ALLOC)
```

also show several important facets of pointer arithmetic. First, a pointer and an integer may be added or subtracted. The construction

```
p + n
```

means the nth object beyond the one p currently points to. This is true regardless of the kind of object p is declared to point at; n is scaled as necessary to make it work.

As a corollary, pointer subtraction is valid: if p and q point to members of the same array, p−q is an integer equal to the number of elements between p and q (plus 1). This fact can be used to write a very efficient pointer version of the function **strlen**, which computes the length of a character string.

```
strlen(s)      /* length of s */
char *s;
{
        char *p = s;

        while (*p)
                p++;
        return(p-s);
}
```

As formal parameters in a function definition,

```
char s[ ];
```

and

```
char *s;
```

are exactly equivalent; which one to write is determined largely by how expressions will be written in the function.

p is initialized to s, that is, to point to the first character. In the **while** loop, each character in turn is examined until the \0 at the end is seen. Since \0 is zero, and since **while** tests only whether the expression is zero, we have omitted the explicit test. It could be written as

```
while (*p != '\0')
```

Since p points to characters, p++ advances p to the next character each time, and p−s gives the number of characters advanced over, that is, the string length. Pointer arithmetic is consistent: if we had been dealing with **float**'s, which occupy more storage than **char**'s, and if p were a pointer to **float**, p++ would still advance to the next **float**.

You might find it instructive to compare this to the version in Chapter 2.

The second aspect of pointer arithmetic is that pointers may be compared under certain circumstances. Again, if p and q point to members of the same array, then relations like <, >=, etc., work properly.

```
p < q
```

is true, for example, if p points to an earlier member of the array than does q. The relations == and != also work. Any pointer can be meaningfully compared for equality or inequality with NULL. Other comparisons may not do what you expect, however. For instance, the innocent-looking code

```
int *p;

p = 1;
if (p == 1)
        printf("hello\n");
```

will *never* print "hello" because the 1 in p==1 is scaled to int before the comparison, and thus isn't 1 any more.

All bets are off if you do arithmetic or comparisons with pointers pointing to different arrays. The best that will happen is that your code will work on one machine but collapse mysteriously on another. The worst is that you'll get nonsense on all machines.

All of these considerations mean that we could write another version of alloc which maintains, let us say, float's instead of char's, merely by changing the declaration

```
char *alloc(n)
```

to

```
float *alloc(n)
```

Since all the pointer manipulations automatically take into account the size of the object pointed to, no other parts of alloc and free have to be altered.

5.5 Character Pointers and Functions

By definition, a "string constant", written as

```
"I am a string"
```

is an array of characters. The compiler terminates the array with the character \0 so that programs can find the end.

Although a variable may not hold a string, it may hold a pointer to one, as in the sequence

```
char *message;

message = "now is the time";
```

This assigns the pointer to the string to **message**, which can be manipulated as desired. Note that this is *not* a string copy; only pointers are involved. C does not provide any operators for processing character strings as a unit.

Since one of the most common uses of pointers is accessing character arrays, we will illustrate some aspects of pointers and arrays by studying three useful functions from the standard I/O library to be discussed in Chapter 7.

The first function is **strcpy(s, t)**, which copies the string **t** to the string **s**. The arguments are written in this order by analogy to assignment, where one would say

```
s = t;
```

to assign **t** to **s**. The array version is reminiscent of **strlen**:

```
strcpy(s, t)  /* copy t to s */
char s[ ], t[ ];
{
        int i;

        i = 0;
        while (s[i] = t[i])
                i++;
}
```

For contrast, here is a first version of **strcpy** with pointers.

```
strcpy(s, t)  /* copy t to s */
char *s, *t;
{
        while (*s = *t) {
                s++;
                t++;
        }
}
```

Because arguments are passed by value, **strcpy** can use **s** and **t** in any way it pleases. Here they are conveniently initialized pointers, which are marched along the arrays a character at a time.

In practice, **strcpy** would not be written as we showed it above; it would almost certainly be

```
strcpy(s, t)   /* copy t to s */
char *s, *t;
{
        while (*s++ = *t++)
                ;
}
```

*t++ is a two step operation, with a side effect. Unary operators like * and ++ operate right to left. Thus in a combination like

```
*t++
```

the ++ operator is done first; this produces the *value* of t, and has the *side effect* of incrementing t. Then the * is applied to the old value of t to access the character t pointed to. In the same manner, this character is assigned to the old position that s pointed to and s is incremented. Finally, the value of the assignment statement, the original character that t pointed to, is used: the character is tested to determine whether the loop should go around again. When the \0 has been safely copied over, the loop will terminate.

Although this may seem unduly complicated, the notational convenience is considerable, and the idiom should be mastered.

The second function is strcat(s, t), which concatenates the string t to the end of s. strcat assumes that s is large enough to hold the combination; if this is not the case, it is possible to use alloc to assign new space for the created string. First, strcat with conventional array indexing, as in Chapter 2.

```
strcat(s, t)   /* concatenate t to end of s */
char s[ ], t[ ];
{
        int i, j;

        i = j = 0;
        while (s[i])   /* find end of s */
                i++;
        while (s[i++] = t[j++])  /* copy t */
                ;
}
```

Note the postfix ++ to increment i and j so they are always ready for the next character.

Now here is strcat with pointers.

```
strcat(s, t)    /* concatenate t to end of s */
char *s, *t;
{
        char *p;

        p = s;
        while (*p)      /* find end of s */
                p++;
        while (*p++ = *t++)     /* copy t */
                ;
}
```

p starts at the first character of s. Each time p is incremented (p++) it moves one character along the array, and eventually reaches the \0 that terminates s. From that point, characters of t are copied, including the \0 that terminates t. The construction

```
while (*p++ = *t++)
```

is identical to that explained with strcpy above.

The final routine is strcmp(s, t), which compares s to t, and returns negative, zero or positive according as s is less than, equal to, or greater than t. Both s and t are character strings.

```
strcmp(s, t)  /* return <0 if s<t, =0 if s==t, >0 if s>t */
char s[ ], t[ ];
{
        int i;

        i = 0;
        while (s[i] == t[i])
                if (s[i++] == '\0')
                        return(0);
        return(s[i] - t[i]);
}
```

The pointer version of strcmp is not as close a parallel as our previous examples.

```
strcmp(s, t)  /* returns <0 if s<t, 0 if s==t, >0 if s>t */
char *s, *t;
{
        while (*s == *t++)
                if (*s++ == '\0')
                        return(0);
        return(*s - *(t-1));
}
```

By the time it is found that the strings differ at some point, t has been

incremented one position too far. To access the previous character, that is, the one before where t currently points, we need

```
*(t-1)
```

Accordingly, the **return** statement is

```
return(*s - *(t-1));
```

Since ++ and -- can occur as either prefix or postfix operators, other combinations of * and ++ and -- occur, although less frequently. For example,

```
*++p
```

increments p *before* fetching the character that p points to;

```
*--p
```

decrements first. This means that we could have written the last **return** statement in **strcmp** as

```
return(*s - *--t);
```

Exercise 5-2: Write the function **strcat** with a call to **alloc** to allocate enough space for the created string. What value should **strcat** return, if any? □

5.6 Pointer Copying

You may notice in older C programs a rather cavalier attitude toward pointer copying. It has generally been true that in C a pointer may be assigned to an integer and back again without changing it. This has led to the taking of liberties with routines that return pointers which are then merely passed to other routines — the requisite pointer declarations are often left out. For example, in

```
if (strcmp(strcat(p, q), strcat(s, t)) == 0) ...
```

there would be a strong tendency not to bother declaring that **strcat** returns a character pointer. In fact, **strcat** itself would probably not be declared as returning a character pointer.

This kind of code is inherently risky, for it depends on details of implementation and machine architecture which may well not hold for the particular compiler you use. It's wiser to be complete in all declarations.

5.7 Two-Dimensional Arrays

Consider the problem of date conversion, from day of the month to day of the year and vice versa. These computations, which would presumably be done by two separate functions, both need the same information, a table of the number of days in each month ("thirty days hath September ..."). Since the number of days per month differs for leap years and non-leap years, it's easiest to separate them into two rows of a two-dimensional table, rather than try to keep track of what happens to February during computation. The table must be shared between two routines: **day_of_year**, which converts the month and day into the day of the year, and **month_day**, which converts the day of the year into the month and day. The table and the functions for performing the transformations are as follows:

```
static int day_tab[2][13]  ={
            {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
            {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}
};

day_of_year(year, month, day)
int year, month, day;
{
        int i, leap;

        leap = (year%400 == 0) || (year%100 != 0 && year%4 == 0);
        for (i = 1; i < month; i++)
                day += day_tab[leap][i];
        return(day);
}

month_day(year, yearday, month, day)
int year, yearday, *month, *day;
{
        int i, leap;

        leap = (year%400 == 0) || (year%100 != 0 && year%4 == 0);
        for (i = 1; yearday > day_tab[leap][i]; i++)
                yearday -= day_tab[leap][i];
        *month = i;
        *day = yearday;
}
```

The table **day_tab** has to be external to both **day_of_year** and **month_day**. Since its definition precedes them both, no **extern** declarations are needed. And since **month_day** has to return a pair of values, it does so by assuming it has been called with two pointers.

The table **day_tab** is the first two-dimensional array we have dealt with. A two-dimensional array is really a one-dimensional array, each of whose

elements is an array. Hence subscripts are written as

```
day_tab[i][j]
```

rather than

```
day_tab[i, j]
```

as in most languages. Other than this, a two-dimensional array can be treated in much the same way as in other languages. Elements are stored by rows.

The array is initialized by a list of initializers in braces; each row is initialized by the corresponding list. We started the array with a column of zero so that month numbers can run from the intuitively natural 1 to 12 instead of 0 to 11. Since space is certainly not at a premium here, this is easier than mentally adjusting indices.

5.8 Pointer Arrays; Pointers to Pointers

Since pointers are variables themselves, you might reasonably expect that there would be a need for arrays of pointers. This is indeed the case. We have already seen a limited example, although we didn't stress it. A two-dimensional array like

```
int     day_tab[2][13]
```

is actually a one-dimensional array `day_tab[2]` with two elements, each of which is a one-dimensional array of 13 elements. And since we have been saying that pointers and arrays are really much the same thing, this means that `day_tab` can be considered an array of two pointers to integers, or a pointer to a pointer to integers.

Let us illustrate this further with a rather larger program than most we have written so far. The task is to write a program that will sort a set of text lines into alphabetic order, a stripped-down version of the Unix utility *sort*.

In Chapter 3 we presented a Shell sort function that would sort an array of integers. The same algorithm will work, except that now we have to deal with lines of text, which are of different sizes, and certainly can't be compared or moved in a single operation as can integers. What data representation will cope efficiently and conveniently with variable-length text lines?

This is where the array of pointers enters. If we store the lines to be sorted end to end in one long character array (maintained by `alloc`, perhaps) then we can refer to each line by a pointer to its first character. The pointers themselves can be stored in an array. Now when two out-of-order lines are to be exchanged, the *pointers* in the pointer array are exchanged, not the text lines themselves. This eliminates the twin problems of complicated storage management and high overhead that would be part of moving the actual lines.

The sorting process involves three steps:

read all the lines of input
sort them
print them in order

As usual, it's best to divide the program into functions that match this natural division, with the main routine controlling things.

Let us defer the sorting step for a moment, and concentrate on the data structure and the input and output. The main item of data structure needed is an array to hold the pointers to the lines as they are read in, and a line count for sorting. The input routine has to collect the lines, fill this pointer array, and count the input lines. Since the input function can only cope with a finite number of input lines, it can return some illegal line count like -1 to signal an input overflow. The output routine only has to print the lines in the order in which they appear in the array of pointers.

Here is the bulk of the code.

```
#define     NULL 0
#define     LINES 100   /* maximum lines to be sorted */
#define     MAXLINE     200   /* longest line to handled */

main( )     /* sort input lines */
{
        char *lineptr[LINES];       /* pointers to text lines */
        int nlines;                 /* number of input lines read */

        if ((nlines = readlines(lineptr)) >= 0) {
                sort(lineptr, nlines);
                writelines(lineptr, nlines);
        }
        else
                printf("input too big to sort\n");
}
```

```
readlines(lineptr)   /* read input lines for sorting */
char *lineptr[ ];
{
        int n, nlines;
        char *p, *alloc( ), line[MAXLINE];

        nlines = 0;
        while ((n = getline(line, MAXLINE)) >= 0)
                if (nlines >= LINES)
                        return(-1);
                else if ((p = alloc(n+1)) == NULL)
                        return(-1);
                else {
                        strcpy(p, line);
                        lineptr[nlines++] = p;
                }
        return(nlines);
}

writelines(lineptr, nlines)  /* write output lines */
char *lineptr[ ];
int nlines;
{
        int i;

        for (i = 0; i < nlines; i++)
                printf("%s\n", lineptr[i]);
}
```

The main new thing is the declaration for **lineptr**,

```
char *lineptr[LINES];       /* pointers to text lines */
```

which says that **lineptr** is an array of **LINES** elements, each a pointer to **char**. That is, **lineptr[i]** is a character pointer, and ***lineptr[i]** accesses a character.

main includes the call to **sort**, but we actually wrote the program originally without any **sort**, just **readlines** and **writelines**, to make sure that we could copy lines from input to output intact, and that the error-detecting parts worked properly. With that all checked out, we could then concentrate on the sorting. Writing a program in small steps — "incremental construction," if you like — is a good approach: when a program falls apart during development, it is almost certainly related to the most recent thing you added. Knowing that much makes it a lot easier to find the trouble.

Now we can proceed to sorting. The Shell sort from Chapter 3 needs only minor changes, mainly new declarations, and separation of the comparison operation into a separate function.

```
sort(v, n)      /* sort v[0] ... v[n-1] into increasing order */
char *v[ ];
int n;
{
        int gap, i, j;
        char *k;

        for (gap = n/2; gap > 0; gap /= 2)
                for (i = gap; i < n; i++)
                        for (j = i-gap; j >= 0; j -= gap) {
                                if (strcmp(v[j], v[j+gap]) <= 0)
                                        break;
                                k = v[j];
                                v[j] = v[j+gap];
                                v[j+gap] = k;
                        }
}
```

The declaration of k is

```
char *k
```

Since any individual element of v (alias lineptr) is a character pointer, k should be the same so one can be copied to the other.

We also wrote the program about as straightforwardly as possible, so as to get it working quickly. It is quite likely that the sort could be faster if, for instance, we copied the incoming lines directly into an array maintained by readlines, rather than copying them into line and then to a hidden place maintained by alloc. But it's wiser to make the first draft something easy to understand; worry about "efficiency" later. The way to make this program significantly faster is probably not by avoiding an unnecessary copy of the input lines. Replacing the Shell sort by something better, like Quicksort, is more likely to make a difference.

Initialization of an array of pointers is straighforward; again, all that is needed is a list of initializers enclosed in braces. One of the most common initializers is a set of character strings, as in this routine to print error messages:

```
char *msg[ ] ={
        "too bad",
        "tough luck",
        "it's all over",
        "so long"
};

print_msg(n)
int n;
{
        printf("error: %s\n", msg[n]);
}
```

The compiler supplies the proper count for the array by counting the initializers.

Exercise 5-3: Rewrite the routines **day_of_year** and **month_day** with pointers instead of indexing. □

5.9 Command Arguments

In most environments that support C, there is a way to specify arguments or parameters to be passed to the program when it begins executing. These arguments are available to the function **main** (if the program wishes to do anything with them) as an argument count **argc** and an array of character strings **argv** containing the arguments. Manipulating these character string arguments is one of the more common uses of multiple levels of pointers.

As the simplest illustration of the necessary declarations and use, here is a program that simply echoes its arguments. By convention, **argc** is greater than zero; the first argument (**argc**=1) in **argv[0]** is the command name itself.

```
main(argc, argv)    /* echo arguments; 1st version */
int argc;
char *argv[ ];
{
        int i;

        for (i = 1; i < argc; i++)
                printf("%s%s", argv[i], (i < argc-1) ? " " : "\n");
}
```

Since the zeroth argument is the command name, we start with the first argument, **argv[1]**. When printed, each argument is a character string in exactly the same way as the lines of text were in the sort program. Each argument except the last is followed by a blank.

Since **argv** is an array of pointers, there are several different ways to write this program. Let us show two others.

```
main(argc, argv)    /* echo arguments; 2nd version */
int argc;
char *argv[ ];
{
        while (--argc > 0)
                printf("%s%s", *++argv, (argc > 1) ? " " : "\n");
}
```

Each time the pointer **argv** is incremented, one more argument is dropped from the "bottom" of the list. At the same time **argc** is decremented; when it becomes zero, there are no arguments left to print.

Alternatively

```
main(argc, argv)    /* echo arguments; 3rd version */
int argc;
char *argv[ ];
{
        while (--argc > 0)
                printf((argc > 1) ? "%s " : "%s\n", *++argv);
}
```

This version shows that the format argument of **printf** can be an expression just like any of the others. This usage is not very frequent, but worth remembering.

As a larger example, let us make some enhancements to the pattern-finding program we wrote in Chapter 4. If you recall, we wired the search pattern deep into the program, an obviously unsatisfactory arrangement. Following the lead of the Unix program *grep*, let us redesign so the pattern to be searched for is specified by the first argument on the command line.

```
#define     MAXLINE     1000

main(argc, argv)    /* search for pattern from first argument */
int argc;
char *argv[ ];
{
        char line[MAXLINE];

        while (getline(line, MAXLINE) >= 0)
                if (index(line, argv[1]) >= 0)
                        printf("%s\n", line);
}
```

The basic model can now be elaborated to illustrate further pointer constructions. Suppose we want two optional arguments. One says "print *all but* the lines that match the pattern;" the second says "precede each printed line with its line number."

A common convention for C programs is that an argument beginning with a minus sign '−' introduces an optional flag or parameter. If we choose −a (for "all but") to signal the inversion, and −n ("number") line numbering, then the command

```
find −a −n the
```

with the input

```
now is the time
for all good men
to come to the aid
of their party.
```

should produce the output

```
2:for all good men
```

Ideally the implementation of the optional arguments should be such that the rest of the program is relatively insensitive to the details of the arguments which were actually present. In particular, we don't want the call to **index** to refer to **argv[2]** when there was a flag argument and to **argv[1]** when there wasn't.

To achieve this requires us to treat **argv** itself as a pointer which can be incremented.

```
#define     MAXLINE    1000

main(argc, argv)    /* search for pattern */
int argc;
char *argv[ ];
{
        char line[MAXLINE];
        int allbut = 0, number = 0, n = 0;

        while (argc > 1 && argv[1][0] == '-') {
                switch (argv[1][1]) {
                        case 'a':       allbut = 1; break;
                        case 'n':       number = 1; break;
                }
                argc--;
                argv++;
        }
        while (getline(line, MAXLINE) >= 0) {
                n++;
                if ((index(line, argv[1]) >= 0) != allbut) {
                        if (number)
                                printf("%d:", n);
                        printf("%s\n", line);
                }
        }
}
```

argv is incremented if an optional argument is found. Since **argv** is a pointer to the beginning of the array of argument strings, incrementing it by 1 makes it point at the original **argv[1]** instead of **argv[0]** as it used to. At the same time we decrement **argc** in case anything further on in the program depends on that. (It doesn't in this case.) We also mentioned earlier that a function could treat arguments as either arrays or pointers, or even both; this example illustrates that.

Exercise 5-4: Write the program **tail** which prints the last *n* lines of its input. By default, *n* is 10, let us say, but it can be changed by an optional argument, so that

```
tail -n
```

prints the last *n* lines, for all reasonable *n*. Write the program so it makes the best use of available storage. □

5.10 Pointers to Functions

We said earlier that the operator & can be applied to functions, which implies that a *pointer to a function* is a legitimate object. In C, a function itself is not a variable, but it is quite possible to define a pointer to a function, which can be manipulated, passed to functions, and so on. This gives essentially the same capabilities as would be possible if functions were variables, but is much simpler to deal with.

Let us illustrate the use of pointers to functions, by modifying the sorting procedure that we wrote earlier in this chapter. A sort consists generally of three steps — a *comparison* which determines the ordering of any pair of objects, an *exchange* which reverses their order, and a sorting algorithm which makes comparisons and exchanges until the objects are in order.

If we use different comparison and exchange functions, we can sort objects any way we please. The sorting algorithm is independent of the comparison and exchange operations, so it need not change.

Let us modify the previous sorting program so the comparison and exchange functions are passed to **sort** as pointers to functions. **sort** in turn will call the functions via the pointers. First, the main routine needs to declare the functions **strcmp** and **swap** (a new routine), and pointers to them must be passed to **sort**:

```
#define     NULL 0
#define     LINES 100   /* maximum lines to be sorted */
#define     MAXLINE   200   /* longest line to handled */

main( )     /* sort input lines */
{
        char *lineptr[LINES];       /* pointers to text lines */
        int nlines;             /* number of input lines read */
        int strcmp( ), swap( );     /* comparison and exchange functions */

        if ((nlines = readlines(lineptr)) >= 0) {
                sort(lineptr, nlines, &strcmp, &swap);
                writelines(lineptr, nlines);
        }
        else
                printf("input too big to sort\n");
}
```

&strcmp and **&swap** are pointers to the functions; in fact, since they are known to be functions, the & is unnecessary.

The second step is to modify **sort**:

```
sort(v, n, comp, exch)      /* sort v[0] ... v[n-1] into increasing order */
char *v[ ];
int n;
int (*comp)( ), (*exch)( );
{
        int gap, i, j;

        for (gap = n/2; gap > 0; gap /= 2)
                for (i = gap; i < n; i++)
                        for (j = i-gap; j >= 0; j -= gap) {
                                if ((*comp)(v[j], v[j+gap]) <= 0)
                                        break;
                                (*exch)(&v[j], &v[j+gap]);
                        }
}
```

The declarations should be studied with some care.

```
int (*comp)( )
```

says that the object in question is a pointer to a function that returns an int. Without the first set of parentheses,

```
int *comp( )
```

would say that comp was a function returning a pointer to an int, which is quite a different thing.

The use of the comparison function in the line

```
if ((*comp)(v[j], v[j+gap]) <= 0)
```

is the consistent with the declaration: *comp is a pointer to a function; it is followed by its argument list.

The final step is to add the function swap which exchanges two pointers. This is of course adapted directly from a routine we presented early in the chapter.

```
swap(px, py) /* swap *px and *py */
char *px[ ], *py[ ];
{
        char *k;

        k = *px;
        *px = *py;
        *py = k;
}
```

Notice that the pointers involve two levels of indirection.